

Scheme-based Theorem Discovery and Concept Invention

O. Montano-Rivas^a, R. McCasland^a, L. Dixon^a, A. Bundy^a

^a*School of Informatics, University of Edinburgh, Informatics Forum, 10 Crichton Street,
Edinburgh EH8 9AB, UK*

Abstract

We describe an approach to automatically invent/explore new mathematical theories, with the goal of producing results comparable to those produced by humans, as represented, for example, in the libraries of the Isabelle proof assistant. Our approach is based on ‘schemes’, which are terms in higher-order logic. We show that it is possible to automate the instantiation process of schemes to generate conjectures and definitions. We also show how the new definitions and the lemmata discovered during the exploration of a theory can be used, not only to help with the proof obligations during the exploration, but also to reduce redundancies inherent in most theory formation systems. We implemented our ideas in an automated tool, called IsaScheme, which employs Knuth-Bendix completion and recent automatic inductive proof tools. We have evaluated our system in a theory of natural numbers and a theory of lists.

Keywords: mathematical theory exploration, schemes, theorem proving,

Email addresses: `O.Montano-Rivas@inf.ed.ac.uk` (O. Montano-Rivas),
`rmccasla@inf.ed.ac.uk` (R. McCasland), `ldixon@inf.ed.ac.uk` (L. Dixon),
`bundy@inf.ed.ac.uk` (A. Bundy)

1. Introduction

Human mathematical discovery processes include the invention of definitions, conjectures, theorems, examples, counter-examples, problems and algorithms for solving these problems. Automating these discovery processes is an exciting area of research with applications to automated theorem proving [1, 2], algorithm synthesis [3], and others [4, 5, 6].

A variety of theory exploration computer programs have been implemented [7, 5, 6] and different approaches have been identified [2]. A recent approach, scheme-based mathematical theory exploration [8], has been proposed and its implementation is being undertaken within the *Theorema* project [9].

A case study of mathematical theory exploration is described in [10] for the theory of natural numbers using the scheme-based approach. However, apart from this paper there is, to our knowledge, no other case study of scheme-based mathematical theory exploration. In the *Theorema* system, which was used to carry out the aforementioned case study, the user had to provide the appropriate substitutions (*Theorema* cannot perform the possible instantiations automatically). The authors also pointed out that the implementation of some provers was still in progress and that the proof obligations were in part ‘pen-and-paper’. From this observation, a natural question arises: can we effectively mechanise the process of selecting the instantiations for schemes, and subsequently automate the proof obligations induced by the conjectures and definitions?

The main contribution of this paper is to give a positive answer to the above question. The scheme-based approach gives a basic facility to instantiate schemes (or rather higher-order variables inside schemes) with different ‘pieces’ of mathematics (terms built on top of constructor and function symbols) already known in the theory. In section 3, we discuss some motivating examples for the generation of conjectures and definitions using schemes. In order to soundly instantiate the schemes, it is necessary to pay attention to the type of objects being instantiated. In sections 4 and 5 we show how this can be performed rigorously and with total automation on top of the simply typed lambda calculus of Isabelle/HOL [11]. To facilitate the process of proof construction, Isabelle provides a number of automatic proof tools. Tools such as the *Simplifier* [12] or *IsaPlanner* [13] can help with the proof obligations for conjectures in the process of theory exploration. Isabelle also has strong definitional packages such as the *function package* [14] that can prove termination automatically for many of the functions that occur in practice. Section 6 shows how new definitions and the lemmata discovered during the exploration of the theory can be used not only to strengthen the aforementioned tools but also to reduce redundancies inherent in most theory formation systems (section 7). In section 8, we describe our theory exploration algorithms where the processes of theorem and definition discovery are linked together. The evaluation is described in section 9. The related and future work are discussed in sections 10 and 11 respectively and the conclusions in section 12.

2. Preliminaries and Conventions

Given a finite set \mathcal{S} of *sort symbols*, and a denumerable set \mathcal{S}^\vee of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\vee}$ of *polymorphic types* is generated from these sets by the constructor \rightarrow for *functional types*. In the following we use τ to denote types and ‘ a ’, ‘ b ’, etc. to denote type variables. The type constructor \rightarrow associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

A *signature* is a set of typed *function symbols* denoted by $\mathcal{F} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^\vee}} \mathcal{F}_\tau$. Terms are generated from a set of typed *variables* $\mathcal{V} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^\vee}} \mathcal{V}_\tau$ and a signature \mathcal{F} where $\mathcal{F} \cap \mathcal{V} = \emptyset$ by λ -abstraction and application, and are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $t :: \tau$ to indicate that the term t has type τ .

We differentiate *free variables* from *bound variables* in that the latter are bound by λ -abstraction. The sets of function symbols and free variables in a term t are denoted by $\mathcal{F}(t)$ and $\mathcal{V}(t)$ respectively. We assume the usual definition of α and β conversion between λ -terms. We follow the convention that terms that are α -congruent are identified (i.e. $\lambda x. x = \lambda y. y$) and unless stated otherwise the variables r, s, t , etc. range over λ -terms in $\beta\eta$ -normal form.

A *substitution* $\sigma :: \mathcal{V} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables into terms of the correct type. Substitutions σ are extended to applications over terms $\hat{\sigma} :: \mathcal{T}(\mathcal{F}, \mathcal{V}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{V})$ in the usual way. To simplify notation we will not distinguish between a substitution σ and its extension $\hat{\sigma}$ and in the following σ will be used to denote both.

A *rewrite rule* is a pair (l, r) of terms such that $l \notin \mathcal{V}$, l and r are of the same type and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. We write $l \rightarrow r$ for (l, r) . A term rewrite system (TRS for short) \mathcal{R} is a set of rewrite rules. The rewrite rules of a TRS

define a reduction relation $\rightarrow_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the usual way. The reflexive transitive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow_{\mathcal{R}}^*$. A term t is reducible iff there is u such that $t \rightarrow_{\mathcal{R}} u$.

3. Motivating examples

The central idea of scheme-based mathematical theory exploration is that of a scheme; i.e. a higher-order formula intended to capture the accumulated experience of mathematicians for discovering new pieces of mathematics. The invention process is carried out through the instantiation¹ of variables within the scheme. As an example, let $\mathcal{T}_{\mathcal{N}}$ be the theory of natural numbers in which we already have the constant function zero (0), the unary function successor (*suc*) and the binary function addition (+) and let s be the following scheme which captures the idea of a binary function defined recursively in terms of other functions.

$$\left(\begin{array}{l} \text{def-scheme}(\mathbf{g}, \mathbf{h}, \mathbf{i}, \mathbf{j}) \equiv \\ \exists f. \forall x y. \quad \wedge \left\{ \begin{array}{l} f(\mathbf{g}, y) = \mathbf{h}(y) \\ f(\mathbf{i}(x), y) = \mathbf{j}(y, f(x, y)) \end{array} \right. \end{array} \right) \quad (1)$$

Here the existentially quantified variable f stands for the new function to be defined in terms of the variables \mathbf{g} , \mathbf{h} , \mathbf{i} and \mathbf{j} . We can generate the definition of multiplication by allowing the theory $\mathcal{T}_{\mathcal{N}}$ to instantiate the scheme with

¹In Theorema, the instantiation process is limited to function, predicate or constant symbols already known in the theory. Additionally, IsaScheme can use any well-formed closed term of the theory including λ -terms such as $(\lambda x. x)$.

$\sigma_1 = \{\mathbf{g} \mapsto 0, \mathbf{h} \mapsto (\lambda x. 0), \mathbf{i} \mapsto \text{suc}, \mathbf{j} \mapsto +\}$ (here $f \mapsto *$).

$$\begin{aligned} 0 * y &= 0 \\ \text{suc}(x) * y &= y + (x * y) \end{aligned}$$

which in turn can be used for the invention of the concept of exponentiation with the substitution $\sigma_2 = \{\mathbf{g} \mapsto 0, \mathbf{h} \mapsto \lambda x. \text{suc}(0), \mathbf{i} \mapsto \text{suc}, \mathbf{j} \mapsto *\}$ on scheme 1 (note that the exponent is the first argument in this case).

$$\begin{aligned} \text{exp}(0, y) &= \text{suc}(0) \\ \text{exp}(\text{suc}(x), y) &= y * \text{exp}(x, y) \end{aligned}$$

Schemes can be used not only for the invention of new mathematical concepts or definitions, they also can be used for the invention of new conjectures about those concepts. The scheme 2 creates conjectures about the *left-distributivity* property of two binary operators in a given theory (the variables \mathbf{p} and \mathbf{q} stand for the binary operators). Therefore if we are working w.r.t. \mathcal{T}_N extended with multiplication and exponentiation, we can conjecture the left-distributivity property of multiplication and addition and also between exponentiation and multiplication by using the substitutions $\sigma_3 = \{\mathbf{p} \mapsto +, \mathbf{q} \mapsto *\}$ and $\sigma_4 = \{\mathbf{p} \mapsto *, \mathbf{q} \mapsto \text{exp}\}$ respectively on the scheme (2).

$$\left(\begin{array}{l} \text{left-distributivity}(\mathbf{p}, \mathbf{q}) \equiv \\ \forall x y z. \mathbf{q}(x, \mathbf{p}(y, z)) = \mathbf{p}(\mathbf{q}(x, y), \mathbf{q}(x, z)) \end{array} \right) \quad (2)$$

The aforementioned substitutions give the conjectures

$$\begin{aligned} x * (y + z) &= (x * y) + (x * z) \\ \text{exp}(x, y * z) &= \text{exp}(x, y) * \text{exp}(x, z) \end{aligned}$$

It is important to note that schemes could generate invalid definitions and false conjectures. For example, consider the substitution $\sigma_4 = \{\mathbf{g} \mapsto 0, \mathbf{h} \mapsto (\lambda x. 0), \mathbf{i} \mapsto (\lambda x. x), \mathbf{j} \mapsto +\}$ ² on scheme 1

$$\begin{aligned} f(0, y) &= 0 \\ f(x, y) &= y + f(x, y) \end{aligned}$$

This instantiation immediately leads to logical inconsistencies by subtracting $f(x, y)$ from the second equation producing $0 = y$. This definition is invalid because, contrary to the natural interpretation of \mathbf{i} as a constructor symbol, schemes do not express such conditions on instantiations. Similarly, we can also obtain false conjectures from a substitution, e.g. $\sigma_4 = \{\mathbf{p} \mapsto *, \mathbf{q} \mapsto +\}$ on scheme 2 instantiates to $x + (y * z) = (x + y) * (x + z)$.

4. Representation of Schemes

A *scheme* is a higher-order formula intended to generate new *definitions* of the underlying theory and *conjectures* about them. However, not every higher-order formula is a scheme. Here, we formally define schemes.

Definition 1. A **scheme** s is a (non-recursive) constant definition of a proposition in HOL which we write in the form $s_n(\bar{x}) \equiv t$.

For the scheme $s_n(\bar{x}) \equiv t$, \bar{x} are free variables and t does not contain s_n , does not refer to undefined symbols and does not introduce extra free variables.

²Note that all theories considered depend upon the simply typed lambda calculus of Isabelle/HOL. Therefore, $(\lambda x. x)$ is a perfectly valid mathematical object. In fact, we can choose to have any (finite) set of well-formed closed terms as the initial theory elements for the exploration of the theory (see section 5 for details).

The scheme (where *dvd* means “divides”) $prime(p) \equiv 1 < p \wedge (dvd(m, p) \Rightarrow m = 1 \vee m = p)$ is flawed because it introduces the extra free variable m on the right hand side. The correct version is $prime(p) \equiv 1 < p \wedge (\forall m. dvd(m, p) \Rightarrow m = 1 \vee m = p)$ assuming that all symbols are properly defined.

Definition 2. Given a scheme $s := s_n(\bar{x}) \equiv t$ we say that s is a **propositional scheme**. In case t has the form $\exists \bar{f} \forall \bar{y} \wedge_{i=1}^m l_i = r_i$ then we say that the propositional scheme s is a **definitional scheme**, and $l_1 = r_1, \dots, l_m = r_m$ are the **defining equations** of s .

Examples of valid propositional schemes are listed below.

$$\begin{aligned} true &\equiv \top \\ comm(p) &\equiv (\forall x y. p(x, y) = p(y, x)) \\ assoc_comm(p) &\equiv \forall x y z. p(p(x, y), z) = p(x, p(y, z)) \wedge comm(p) \end{aligned}$$

The following are examples of definitional schemes.

$$\left(\begin{array}{l} def_scheme(\mathbf{g}, \mathbf{h}, \mathbf{i}, \mathbf{j}) \equiv \\ \exists f. \forall x y z. \wedge \left\{ \begin{array}{l} f(\mathbf{g}, y) = y \\ f(\mathbf{h}(z, x), y) = \mathbf{i}(\mathbf{j}(z, y), f(x, y)) \end{array} \right. \end{array} \right) \quad (3)$$

$$\left(\begin{array}{l} mutual_def_scheme(\mathbf{g}, \mathbf{h}, \mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{l}) \equiv \\ \exists f_1 f_2. \forall x y. \wedge \left\{ \begin{array}{l} f_1(\mathbf{g}) = \mathbf{h} \\ f_2(\mathbf{g}) = \mathbf{i} \\ f_1(\mathbf{j}(z, x)) = \mathbf{k}(z, f_2(x)) \\ f_2(\mathbf{j}(z, x)) = \mathbf{l}(z, f_1(x)) \end{array} \right. \end{array} \right) \quad (4)$$

The definitional scheme (4) captures the idea of two mutual functions defined recursively. Here the existentially quantified variables (f in scheme (3) and f_1 and f_2 in scheme (4)) stand for the new functions to be defined.

5. Generation of Instantiations

In this section we describe the technique used to instantiate schemes automatically. Here we define some preliminary concepts.

Definition 3. *For a scheme s , the set of **schematic substitutions** with respect to a (finite) set of closed terms $X \subset \mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined by:*

$$Sub(s, X) := \{\sigma \mid Closed(s\sigma) \wedge Ran(\sigma) \subseteq X\}$$

where $Closed(t)$ is true when the term t contains no free variables, $Ran(\sigma) := \{x\sigma \mid x \in Dom(\sigma)\}$ and $Dom(\sigma) := \{x \in \mathcal{V} \mid x\sigma \neq x\}$.

Ensuring that $s\sigma$ is a closed term helps avoid over generalisations of conjectures or definitions, e.g. it is impossible to prove $\forall x y z. x * \mathbf{p}(y, z) = \mathbf{p}(x * y, x * z)$ where \mathbf{p} is free. Definition 3 also restricts substitutions to be within X . The problem of finding the substitutions $Sub(s, X)$ of a scheme s given a set of terms X can be solved as follows. The free variables $\mathcal{V}(s) = \{v_1, \dots, v_n\}$ in the scheme are associated with their initial domain $D_{0i} = \{x \in X \mid v_i \text{ and } x \text{ can be unified}\}$ for $1 \leq i \leq n$. The typing information of the partially instantiated scheme is the only constraint during the instantiation of variables. Each time a variable v_i is instantiated to $x \in D_{ki}$ the domains $D_{(k+1)j}$ for $i < j \leq n$ of the remaining variables must be updated w.r.t the most general unifier σ_{mgu} of v_i and x . Variables are instantiated sequentially and if a partial instantiation leaves no possible values for a variable then backtracking is performed to the most recently instantiated variable that still has alternatives available. This process is repeated using backtracking to exhaust all possible schematic substitutions obtaining a complete algorithm.

Example 1. Let \mathcal{F} be a signature consisting of $\mathcal{F} := \{+::\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, *::\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, @::\text{a list} \rightarrow \text{a list} \rightarrow \text{a list}, \text{map}::(\text{a} \rightarrow \text{b}) \rightarrow \text{a list} \rightarrow \text{b list}\}$. Also let $X = \{+, *, @, \text{map}\}$ and s be the propositional scheme (2) of section 3 (here we assume the most general type inferred for the scheme).

Figure (1) illustrates how $\text{Sub}(s, X)$ is evaluated following a sequential instantiation of the free variables of s . It is important to note that the asymptotic running time of the algorithm is $\Theta(|X|^{|\mathcal{V}(s)|})$ and the worst case is when we obtain $|X|^{|\mathcal{V}(s)|}$ valid substitutions.

For a scheme s , the generated schematic substitutions are used to produce instantiations of s ; i.e. conjectures or definitions

Definition 4. Given $\sigma \in \text{Sub}(s, X)$, the **instantiation of the scheme** $s := u \equiv v$ with σ is defined by

$$\text{inst}(u \equiv v, \sigma) := v\sigma$$

Definition 5. For a scheme s , the **set of instantiations** $\text{Insts}(s, X)$ with respect to a (finite) set of closed terms $X \subset \mathcal{T}(\mathcal{F}, \mathcal{V})$ is denoted by

$$\text{Insts}(s, X) := \{\text{inst}(s, \sigma) \mid \sigma \in \text{Sub}(s, X)\} \quad (5)$$

Example 2. The instantiations generated from scheme (2) and the set of terms $X = \{+, *, @, \text{map}\}$ are depicted in the following table.

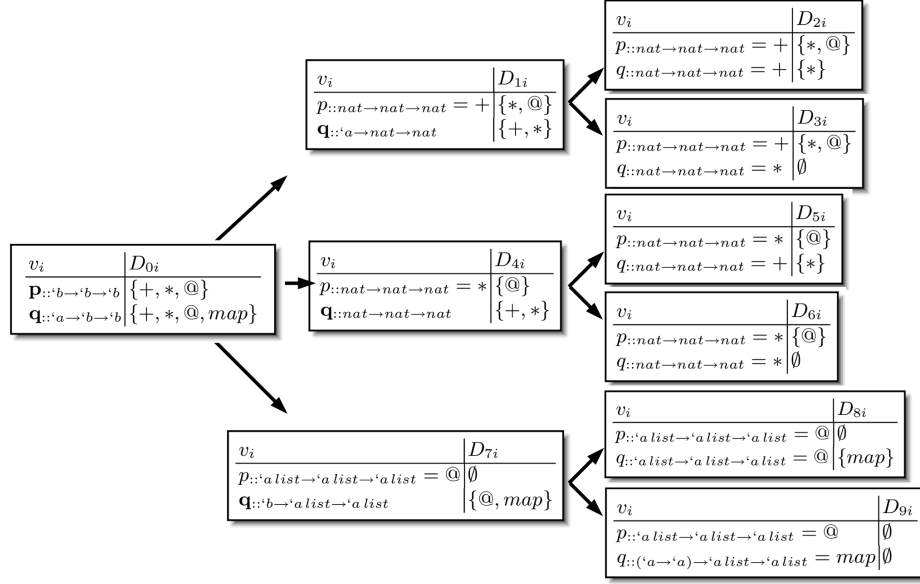


Figure 1: Sequential evaluation of $Sub(s, X)$ where s is the propositional scheme (2) and $X = \{+::nat \rightarrow nat \rightarrow nat, *::nat \rightarrow nat \rightarrow nat, @::a\ list \rightarrow a\ list \rightarrow a\ list, map::(a \rightarrow b) \rightarrow a\ list \rightarrow b\ list\}$. Each box shows the unified and not-unified (in bold) variables and their domain during the evaluation. The output of the algorithm is the set of substitutions $\{\sigma_1 = \{p \mapsto +, q \mapsto +\}, \sigma_2 = \{p \mapsto +, q \mapsto *\}, \sigma_3 = \{p \mapsto *, q \mapsto +\}, \sigma_4 = \{p \mapsto *, q \mapsto *\}, \sigma_5 = \{p \mapsto @, q \mapsto @\}, \sigma_6 = \{p \mapsto @, q \mapsto map\}\}$. Note that a unified variable potentially changes the types of the rest of the variables, restricting their domain.

$Sub(s, X)$	$Insts(s, X)$
$\sigma_1 = \{p \mapsto +, q \mapsto +\}$	$\forall x y z. x + (y + z) = (x + y) + (x + z)$
$\sigma_2 = \{p \mapsto +, q \mapsto *\}$	$\forall x y z. x * (y + z) = x * y + x * z$
$\sigma_3 = \{p \mapsto *, q \mapsto +\}$	$\forall x y z. x + y * z = (x + y) * (x + z)$
$\sigma_4 = \{p \mapsto *, q \mapsto *\}$	$\forall x y z. x * (y * z) = (x * y) * (x * z)$
$\sigma_5 = \{p \mapsto @, q \mapsto @\}$	$\forall x y z. x@(y@z) = (x@y)@(x@z)$
$\sigma_6 = \{p \mapsto @, q \mapsto map\}$	$\forall x y z. map(x, y@z) = map(x, y)@map(x, z)$

6. Identification of Equivalent Instantiations

Processing the instantiations (conjectures and definitions) of a scheme could be a demanding task. In the worst case, the number of substitutions $\sigma : V \rightarrow X$ is $|X|^{|V|}$. However, we can reduce the number of conjectures and definitions by noticing that two different substitutions σ_1 and σ_2 could lead to equivalent instantiations.

Table 1 shows the set of instantiations $Insts(s, X)$ obtained from the following definitional scheme.

$$\left(\begin{array}{l} def\text{-}scheme(\mathbf{g}, \mathbf{h}, \mathbf{i}) \equiv \\ \exists f. \forall x y. \wedge \left\{ \begin{array}{l} f(\mathbf{g}, y) = \mathbf{h}(\mathbf{g}, \mathbf{g}) \\ f(suc(x), y) = \mathbf{i}(y, f(x, y)) \end{array} \right. \end{array} \right) \quad (6)$$

In Table 1 $inst(s, \sigma_{N1})$ and $inst(s, \sigma_{N2})$ are clearly equivalent³. The key ingredient for automatically detecting equivalent instantiations is a term rewrite system (TRS) R . This rewrite system is used to normalise terms such that only irreducible terms are considered [15].

However, for this idea to work, the constructed TRS R must have the property of being *terminating*. All functions in Isabelle/HOL are terminating to prevent inconsistencies. Therefore, the defining equations for a newly introduced function symbol can be used as a normalising TRS. Furthermore, if we are to include a new equation e to the rewrite system \mathcal{R} during the exploration of the theory then we must prove termination of the extended rewrite system $\mathcal{R} \cup \{e\}$. To this end, we use the termination checker AProVE [16] along with Knuth-Bendix completion to obtain a convergent rewrite system,

³Note that ‘+’ denotes standard addition of naturals.

$Sub(s, X)$	$Insts(s, X)$
$\sigma_{N1} = \left\{ \begin{array}{l} g \mapsto 0, h \mapsto + \\ i \mapsto + \end{array} \right\}$	$\exists f. \forall x y. \wedge \left\{ \begin{array}{l} f(0, y) = 0 + 0 \\ f(suc(x), y) = y + f(x, y) \end{array} \right\}$
$\sigma_{N2} = \left\{ \begin{array}{l} g \mapsto 0, h \mapsto (\lambda x y. x) \\ i \mapsto + \end{array} \right\}$	$\exists f. \forall x y. \wedge \left\{ \begin{array}{l} f(0, y) = 0 \\ f(suc(x), y) = y + f(x, y) \end{array} \right\}$
$\sigma_{N3} = \left\{ \begin{array}{l} g \mapsto 0, h \mapsto + \\ i \mapsto (\lambda x y. x) \end{array} \right\}$	$\exists f. \forall x y. \wedge \left\{ \begin{array}{l} f(0, y) = 0 + 0 \\ f(suc(x), y) = y \end{array} \right\}$
$\sigma_{N4} = \left\{ \begin{array}{l} g \mapsto 0, h \mapsto (\lambda x y. x) \\ i \mapsto (\lambda x y. x) \end{array} \right\}$	$\exists f. \forall x y. \wedge \left\{ \begin{array}{l} f(0, y) = 0 \\ f(suc(x), y) = y \end{array} \right\}$

Table 1: Redundant definitions generated from the definitional scheme 6. Note that the instantiations $inst(s, \sigma_{N1})$ and $inst(s, \sigma_{N2})$ are equivalent as $0+0$ can be ‘reduced’ (within the theory) to 0. $inst(s, \sigma_{N3})$ and $inst(s, \sigma_{N4})$ are similarly equivalent.

if possible (using a similar approach to [17]). The following definition will help with the description of the algorithm for theory exploration of section 8.

Definition 6. *Given a terminating rewrite system \mathcal{R} and an instantiation $i \in Insts(s, X)$ of the form $\forall \bar{x}. s = t$, the **normalising extension** $ext(\mathcal{R}, i)$ of \mathcal{R} with i is denoted by*

$$ext(\mathcal{R}, i) := \begin{cases} \mathcal{R}' & \text{if Knuth-Bendix completion succeeds for} \\ & \mathcal{R} \cup \{s = t\} \text{ with a convergent system } \mathcal{R}' \\ \mathcal{R} \cup \{r\} & \text{if termination succeeds for } \mathcal{R} \cup \{r\} \\ & \text{with } r \in \{s = t, t = s\} \\ \mathcal{R} & \text{otherwise} \end{cases}$$

7. Filtering of Conjectures and Definitions

As suggested by definition 6, IsaScheme updates the rewrite system \mathcal{R} each time a new equational theorem is found. It is thus useful to consider the notion of equivalence of instantiations modulo \mathcal{R} .

Definition 7. *Let u and v be two instantiations and \mathcal{R} a terminating rewrite system. **Equivalence of instantiations modulo \mathcal{R}** is denoted as*

$$u \approx_{\mathcal{R}} v := (\hat{u} =_{\alpha} \hat{v})$$

where \hat{u} and \hat{v} are normal forms (w.r.t. \mathcal{R}) of u and v respectively and $=_{\alpha}$ is term equivalence up to variable renaming.

The problem of identifying equivalent instantiations is just an instance of the so-called *word problem*⁴. This word problem is in general undecidable. Definition 7 tries to solve instances of it through normalisation. However, even if \mathcal{R} is convergent, two equivalent terms can have distinct normal forms. Consider for example, the following defining equations for addition

$$\begin{aligned} 0 + y &= y \\ \text{suc}(x) + y &= \text{suc}(x + y) \end{aligned} \tag{7}$$

Clearly the equations (7) form a convergent rewrite system but the latter would fail to decide equivalence of the terms $x + \text{suc}(y)$ and $\text{suc}(x + y)$ which are already in normal form w.r.t. (7). The problem is that rewriting

⁴Given a set of identities \mathcal{R} and two terms s and t , is it possible to transform the term s into the term t using the equations in \mathcal{R} in both directions?

is not sufficient for inductive theories; here proofs by induction are often required[18].

The aforementioned problem leads to redundancies in the construction of definitions. It is thus useful to consider the notion of equivalence of definitions.

Definition 8. *Let $f, g \in \mathcal{F}$ be two function symbols with type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau_0$. We say that f and g are **equivalent definitions** iff*

$$\forall x_1 \dots x_n. f(x_1, \dots, x_n) = g(x_1, \dots, x_n)$$

Since the exploration process could generate a substantial number of definitions and each of them could potentially produce a multitude of conjectures it becomes necessary to restrict the search space in some way. For example, instead of generating $f_1(x, y) = x^2$ and $f_2(x, y) = y^2$ it would be better to just generate $f(x) = x^2$ and construct f_1 and f_2 on top of f , e.g. $(\lambda x y. f(x))$ and $(\lambda x y. f(y))$. Preliminary studies suggested that a significant proportion of functions synthesised by schemes would ignore one or more of their arguments. Such argument neglecting functions can always be defined with another function using fewer arguments and a λ -abstraction. This motivates the following definition, which we use in section 8, to avoid generating such definitions.

Definition 9. *An **argument neglecting function** $f \in \mathcal{F}$ with type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau_0$, where τ_0 is a base type and $n > 0$, is a function such that*

$$f(x_1, \dots, x_{k-1}, y, x_{k+1}, \dots, x_n) = f(x_1, \dots, x_{k-1}, z, x_{k+1}, \dots, x_n)$$

for some k where $1 \leq k \leq n$.

8. Theory Exploration Algorithms

8.1. Scheme-based Conjecture Synthesis.

The overall procedure for the generation of theorems is described by the pseudocode of `InventTheorems`. The algorithm receives as arguments a set of terms I (conjectures), a terminating rewrite system \mathcal{R} , a set of terms T (theorems), a set of propositional schemes S_p and a set of closed terms X_p from which schemes are to be instantiated. Note that initially, $I = T = \emptyset$.

```
InventTheorems( $I, \mathcal{R}, T, S_p, X_p$ )
1  for each  $i \in \bigcup_{s \in S_p} Insts(s, X_p)$ 
2     $\hat{i} :=$  a normal form of  $i$  w.r.t.  $\mathcal{R}$ 
3    if  $\hat{i}$  is not subsumed by  $T \cup \{True\}$  and
4    there is not a  $j \in I$  such that  $j \approx_R \hat{i}$  and
5    cannot find a counter-example of  $\hat{i}$  then
6      if can prove  $\hat{i}$  then
7         $T := T \cup \{\hat{i}\}$ 
8        if  $\hat{i}$  is of the form  $\forall \bar{x}. s = t$  then
9           $\mathcal{R} := ext(\mathcal{R}, \hat{i})$ 
10    $I := I \cup \{\hat{i}\}$ 
11 return  $\langle I, \mathcal{N}, T \rangle$ 
```

The algorithm iterates through all instantiations obtained from schemes in S_p and the terms X_p . Each instantiation is normalised w.r.t. \mathcal{R} in line 2. Line 3 ensures that the instantiation is neither subsumed by any previously proved theorem nor trivially proved by simplification. Equivalent

instantiations modulo \mathcal{R} are identified in line 4. This line can be implemented efficiently using discrimination nets (or by a constraint mechanism as in [15]) and avoids counterexample checking equivalent instantiations modulo \mathcal{R} . Falsifiable instantiations are detected in line 5 to avoid any proof attempt on such conjectures. Isabelle/HOL provides the counter-example checkers *Quickcheck* [19] and *Nitpick* [20] which are used to refute the false conjectures in the implementation of *IsaScheme*. In case the conjecture is not rejected by the inspection in lines 3, 4 or 5 then a proof attempt is performed in line 6. The prover used for the proof obligations in *IsaScheme* was the automatic inductive prover *IsaPlanner* [13] which implements the *rippling* heuristic [21]. Following a successful proof attempt, line 7 adds the theorem into the set of theorems T . Lines 8 and 9 will update \mathcal{R} with its normalising extension if the theorem proved is an equation.

8.1.1. Irreducibility of \mathcal{R}

Collecting specialised versions of theorems can be unwieldy and does not add anything new to the discovery process. As a consequence of line 3 of the `InventTheorems` algorithm, whenever we prove a theorem t , t is not subsumed by a previously proved theorem. However, t may be a generalisation of a previously proved one. Completion in definition 6, handles this situation returning always an *irreducible*⁵ rewrite system. However, Knuth-Bendix completion does not always succeed and termination checking itself cannot discard specialised versions of proved theorems. The following algorithm can be used instead of simple termination checking in definition 6. It will try to

⁵A TRS \mathcal{R} is **irreducible** if for any rule $r \in R$, r is a normal form in $\mathcal{R} \setminus \{r\}$.

transform a non-irreducible rewrite system into an irreducible one.

```

    Irreducible( $\mathcal{R}'$ ,  $\mathcal{R}$ )
1  if  $\mathcal{R}$  is terminating then
2     $\mathcal{R}'' := \{\hat{r} \mid \exists r \in \mathcal{R}. r \xrightarrow{*}_{\mathcal{R} \setminus \{r\}} \hat{r}\}$ 
3    if  $\mathcal{R} \equiv \mathcal{R}''$  then return some  $\mathcal{R}$ 
4    else Irreducible(some  $\mathcal{R}$ ,  $\mathcal{R}'' \setminus \{True\}$ )
5  else  $\mathcal{R}'$ 

```

As expected, the algorithm `Irreducible` is not complete. However, it often succeeds in practice. The second case of definition 6 can now be lifted to obtain a potentially irreducible rewrite system \mathcal{R}' if `Irreducible(none, $\mathcal{R} \cup \{r\}$)` succeeds with *some* R' where $r \in \{s = t, t = s\}$.

8.1.2. Unfalsified and Unproved Conjectures

Theorems about inductively defined data structures and recursive definitions usually require induction to prove them. Inductive proof is, in general, undecidable and the failure of cut elimination for inductive theories implies that new lemmas or generalisations are often needed [18]. The requirement of new lemmas or generalisations suggests that whenever we prove a new theorem we increase the chances to succeed in a previously failed prove attempt.

The `InventTheorems` algorithm does not try to prove a previously failed proof attempt. However, this is not difficult to implement and for the sake of clarity we did not make explicit delayed proof obligations in the `InventTheorems` algorithm.

8.2. Scheme-based Definition Synthesis.

The generation of definitions is described by the pseudocode of the algorithm `InventDefinitions`. The algorithm takes as input the same arguments received by the `InventTheorems` method. Additionally, it also takes a set of function symbols \mathcal{F} in the current theory, a set of terms D (definitions), a set of definitional schemes S_d and a set of closed terms X_d from which definitional schemes are to be instantiated. Again, initially $D = \emptyset$.

```

    InventDefinitions( $I, \mathcal{R}, T, S_p, X_p, \mathcal{F}, D, S_d, X_d$ )
1  for each  $d \in \bigcup_{s \in S_d} Insts(s, X_d)$ 
2     $\hat{d} :=$  a normal form of  $d$  w.r.t.  $\mathcal{R}$ 
3    let  $\exists f_1 \dots f_n. \forall \bar{y}. e_1 \wedge \dots \wedge e_m = \hat{d}$ 
4    create function symbols  $f'_1, \dots, f'_n$  such that  $f'_i \notin \mathcal{F}$ 
5     $\sigma := \{f_1 \mapsto f'_1, \dots, f_n \mapsto f'_n\}$ 
6     $[e'_1, \dots, e'_m] := [\sigma(e_1), \dots, \sigma(e_m)]$ 
7    if there is not a  $j \in D$  such that  $j \approx_R \hat{d}$  and
8     $[e'_1, \dots, e'_m]$  is well-defined and
9     $f'_1, \dots, f'_n$  are not equivalent to previous functions and
10    $f'_1, \dots, f'_n$  are not argument neglecting then
11      $\mathcal{F} := \mathcal{F} \cup \{f'_1, \dots, f'_n\}$ 
12      $T := T \cup \{e'_1, \dots, e'_m\}$ 
13      $\mathcal{R} := \mathcal{R} \cup \{e'_1, \dots, e'_m\}$ 
14      $\langle I, \mathcal{N}, T \rangle := InventTheorems(I, \mathcal{R}, T, S_p, X_p \cup \{f'_1, \dots, f'_n\})$ 
15      $D := D \cup \{\hat{d}\}$ 
16 return  $\langle I, \mathcal{R}, T, \mathcal{F}, D \rangle$ 

```

The algorithm iterates through all instantiations obtained from definitional schemes in S_d and terms in X_d . Each instantiation d is reduced to a normal form \hat{d} w.r.t. \mathcal{R} in line 2. Since \hat{d} is generated from a definitional scheme, it has the form $\exists f_1 \dots f_n. \forall \bar{y}. e_1 \wedge \dots \wedge e_m$ where f_1, \dots, f_n are variables standing for the new functions to be defined and e_1, \dots, e_m are the defining equations of the functions. In lines 4 and 5, new function symbols f'_1, \dots, f'_n (w.r.t. the signature \mathcal{F}) are created and a substitution σ is constructed to uniquely denote each of the new functions to be defined. This ‘renaming’ of functions is performed with the defining equations and $[e'_1, \dots, e'_m]$ is obtained in line 6. Line 7 ensures that definitions that are equivalent modulo \mathcal{R} to earlier generated ones, are ignored. Well-definedness properties, such as termination and totality of the functions generated, are proved in line 8. We used Isabelle/HOL’s *function package* [14] for these proof obligations. Definitions that are identified to be equivalent, by theorem proving, to previously defined functions are rejected in line 9. Line 10 checks if the new functions created are not argument neglecting (AN). Prior to any proof attempt at the conjectures demanded by lines 9 and 10 (definitions 8 and 9), we try to produce counter-examples of those functions being equivalent or AN. In case an instantiation \hat{d} is not rejected by lines 7, 8, 9 or 10, then the context \mathcal{F} and the theorems T are updated with the new function symbols f'_1, \dots, f'_n and the theorems $\{e'_1, \dots, e'_m\}$ respectively (lines 11 and 12). Line 13 updates the rewrite system \mathcal{R} with the newly introduced defining equations e'_1, \dots, e'_m . A call to `InventTheorems` is performed in line 14 updating I, \mathcal{N} and T . At the end of each iteration, the instantiation \hat{d} is added to the set of processed definitions D in line 15. Finally, when all instantiations $d \in \bigcup_{s \in S_d} Insts(s, X_d)$

have been processed the values $\langle I, \mathcal{N}, T, \mathcal{F}, D \rangle$ are returned.

8.2.1. *Delaying the Rejection of Definitions*

In general, it is difficult and expensive to prove the conjectures demanded by definitions 8 and 9, especially if we do not know anything about the new function to be analysed. However, analogously to section 8.1.2, we improve the chances to succeed in these proof obligations after the exploration of line 14 of the `InventDefinitions` algorithm.

In the implementation of `IsaScheme`, we delayed tackling the proof obligations of lines 9 and 10, demanded by definitions 8 and 9 respectively, in case of a previously failed prove attempt.

9. Evaluation

We conducted several case studies in the theory of natural numbers and the theory of lists to evaluate how similar were the results obtained with our method and implementation to those in the libraries of the Isabelle proof assistant. We performed a precision/recall analysis with Isabelle's libraries as reference to evaluate the quality of the theorems and definitions found by the `InventTheorems` and `InventDefinitions` algorithms respectively. We kept track of elapsed time, conjectures and definitions synthesised, conjectures proved and not proved. For the evaluation we used a computer cluster where each theory exploration was run in a GNU/Linux node with 2 dual core CPUs and 4GB of RAM memory. We also used Isabelle/2009-2, `IsaPlanner` svn version 2723 and `AProVE` 1.2.

9.1. Natural Numbers

The evaluation of the `InventDefinitions` algorithm was performed with a theory consisting of one datatype for the naturals. We used the following definitional scheme for the analysis

$$\left(\begin{array}{l} \text{def-scheme-binary}(\mathbf{g}, \mathbf{h}, \mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{l}) \equiv \\ \exists f. \forall x y z. \wedge \left\{ \begin{array}{l} f(\mathbf{g}, y) = \mathbf{h}(y) \\ f(\mathbf{i}(z, x), y) = \mathbf{j}(\mathbf{k}(x, y, z), f(x, \mathbf{l}(z, y))) \end{array} \right. \end{array} \right) \quad (8)$$

along with the propositional scheme (9) and the closed terms

$$X_P = \left\{ \begin{array}{l} (\lambda x y. x), (\lambda x y. y), (\lambda x. \text{suc}), \\ (\lambda x y. \text{suc}(x)), (\lambda x y. 0), (\lambda x y. \text{suc}(0)), \end{array} \right\}$$

$$X_D = \left\{ \begin{array}{l} (\lambda x. x), (\lambda x y. x), (\lambda x y. y), (\lambda x y z. x), (\lambda x y z. y), (\lambda x y z. z), \\ (\lambda x. 0), (\lambda x y. 0), (\lambda x. \text{suc}(0)), 0, (\lambda x. \text{suc}), \text{suc} \end{array} \right\}.$$

Notice that while the user has to provide these terms, the above are formulaic in nature. They consist of projections and constructors symbols.

During this exploration round, `IsaScheme` found 16 new functions and two equivalent definitions for addition. However, the standard version in Appendix A.1 was rejected because it was synthesised after \oplus defined as:

$$\begin{aligned} 0 \oplus y &= 0 \\ \text{suc}(x) \oplus y &= x \oplus \text{suc}(y). \end{aligned}$$

We further explored the theory by adding \oplus to the sets X_P and X_D . During this exploration round, `IsaScheme` ended up with 31 functions. Multiplication \otimes was found by `IsaScheme` in this exploration round, defined as:

$$\begin{aligned} 0 \otimes y &= 0 \\ \text{suc}(x) \otimes y &= y \oplus (x \otimes y). \end{aligned}$$

Again, we further explored the theory by adding \oplus and \otimes to the sets X_P and X_D . At this exploration round, IsaScheme ended up with 39 functions (including exponentiation) and proved 315 theorems leaving 56 unfalsified and unproved conjectures. For the theory of natural numbers, IsaScheme obtained a precision of 8% and a recall of 100%. The results for the evaluation of definitions are summarised in table 2.

We evaluated the `InventTheorems` algorithm with a theory consisting of one datatype, of the naturals, and the usual recursive functions for addition, multiplication and exponentiation (which can be found in Appendix A.1). For the analysis of this theory we used the following propositional scheme

$$\left(\begin{array}{l} \text{prop-scheme-binary}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}, \mathbf{t}, \mathbf{u}) \equiv \\ \forall x y z. \mathbf{p}(\mathbf{q}(x, y), \mathbf{r}(x, z)) = \mathbf{s}(\mathbf{t}(x, z), \mathbf{u}(y, z)) \end{array} \right) \quad (9)$$

and the closed terms

$$X_P = \left\{ \begin{array}{l} (\lambda x y. x), (\lambda x y. y), (\lambda x. \text{suc}), (\lambda x y. \text{suc}(x)), \\ (\lambda x y. 0), (\lambda x y. \text{suc}(0)), +, *, ^ \end{array} \right\}$$

IsaScheme produced a total of 23 theorems for the theory of naturals with 16 of them included in Isabelle’s libraries. Isabelle contains 35 theorems about addition, multiplication and exponentiation giving a precision of 70% and a recall of 46%. The theorems discovered in the theory of natural numbers can be found in table B.4 and included, commutativity, associativity, distributivity of multiplication over addition, distributivity of exponentiation over multiplication, commuted versions of addition and multiplication, among others.

There are 19 theorems in Isabelle not synthesised by IsaScheme. But the following 16 of them are all subsumed by the theorems in table B.4 after

normalisation w.r.t. the rewrite system \mathcal{R}

$$\begin{array}{ll}
(a * m) + m = (a + \text{suc}(0)) * m & a + (c + d) = (a + c) + d \\
m + a * m = (a + \text{suc}(0)) * m & (a + b) + c = (a + c) + b \\
m + m = (\text{suc}(0) + \text{suc}(0)) * m & a * \text{suc}(0) = a \\
(lx * ly) * (rx * ry) = lx * (ly * (rx * ry)) & \text{suc}(0) * a = a \\
lx * (rx * ry) = (lx * rx) * ry & (x \hat{ } q) * x = x \hat{ } \text{suc}(q) \\
(lx * ly) * rx = (lx * rx) * ly & x * (x \hat{ } q) = x \hat{ } \text{suc}(q) \\
x \hat{ } (\text{suc}(\text{suc}(0)) * n) = (x \hat{ } n) * (x \hat{ } n) & x * x = x \hat{ } (\text{suc}(\text{suc}(0))) \\
x \hat{ } (\text{suc}(\text{suc}(\text{suc}(0)) * n)) = x * ((x \hat{ } n) * (x \hat{ } n)) & x \hat{ } \text{suc}(0) = x
\end{array}$$

and the other three ones fell out of the scope of the propositional scheme used as they contained 4 variables (the results are summarised in table 3), in particular they are:

$$\begin{array}{l}
(lx * ly) * (rx * ry) = (lx * rx) * (ly * ry) \\
(lx * ly) * (rx * ry) = rx * ((lx * ly) * ry) \\
(a + b) + (c + d) = (a + c) + (b + d).
\end{array}$$

9.2. Lists

The evaluation of the `InventDefinitions` algorithm was performed with a theory consisting of one datatype for the lists. We used the propositional scheme (9), the definitional scheme (8) and the closed terms

$$\begin{array}{l}
X_P = \left\{ (\lambda x y. x), (\lambda x y. y), (\lambda x y. []), \#, (\lambda x y. y \# x) \right\} \\
X_D = \left\{ (\lambda x. x), (\lambda x y. x), (\lambda x y. y), (\lambda x y z. x), (\lambda x y z. y), \right. \\
\left. (\lambda x y z. z), (\lambda x. []), [], (\lambda x. \#), \#, (\lambda x y. y \# x) \right\}.
\end{array}$$

IsaScheme produced 24 definitions for the theory of lists with 6 of them included in Isabelle's libraries. Isabelle contains 29 recursive functions in

Precision-Recall	8%-100%	25%-21%
Constructors	Z, S	N, C
Functions Symbols	\oplus, \otimes	
Elapsed Time (s)	32414	7575
Conjectures Synthesised	2241013	325151
Conjectures Filtered	2240652	325101
Proved-Not Proved	315-56	28-22
Definitions Synthesised	75059	11563
Definitions not Rejected	39	24

Table 2: Precision and recall analysis for definition synthesis with Isabelle’s theory library as reference. The constructors are 0, *suc*, [] and # with labels Z, S, N and C respectively.

its `List` theory giving a precision of 25% and a recall of 21%. The definitions synthesised by IsaScheme were `append`, (binary version of) `head` (*hd*), `map`, (binary version of) `last` (*last*), tail recursive reverse (*qrev*) and replicate (*replicate*), among others. A representative selection of definitions synthesised are shown in Appendix A.

The low recall for the list theory was caused because the definitional scheme used was only able to synthesise binary functions and not unary or ternary ones. This could have been addressed easily by considering definitional schemes producing unary and ternary functions at the expense of computational time (see section 5). Note that the scheme-based approach for the generation of definitions provides a free-form incremental construction of (potentially infinitely many) recursive functions. Overly general definitional schemes provide a wide range of possible instantiations and thus, definitions.

In fact, we believe this was the reason of the low precision in the evaluation of the algorithm for both theories. Strategies to assess the relevance of definitions are required given the big search space during exploration. This is left as future work. For space reasons we cannot give a presentation of the theories or the theorems and definitions found. However, formal theory documents in human-readable Isabelle/Isar notation and all results described in this paper are available online⁶.

We evaluate the `InventTheorems` algorithm using two different theories of lists. The definitions of the functions used in these theories can be found in Appendix A. The first theory consisted of `append` (`@`), `reverse` (`rev`), `map` (`map`) and `length` (`len`). For the analysis of this theory we used the propositional scheme (9) with the terms

$$X_P = \left\{ \begin{array}{l} (\lambda x y. x), (\lambda x y. y), (\lambda x. []), \#, @, (\lambda x y. y @ x), (\lambda x y. rev(x)) \\ (\lambda x y. rev(y)), (\lambda x y. len(x)), (\lambda x y. len(y)), map, (\lambda x y. map(y, x)) \end{array} \right\}$$

IsaScheme produced a total of 9 theorems for this theory, including all 7 theorems about `@`, `rev`, `map` and `len` (see table B.5). This gives a precision of 77% and a recall of 100%. The extra theorems synthesised were $len(z @ x) = len(x @ z)$ and $len(map(y, x)) = len(x)$.

The second theory analysed consisted of the functions `append` (`@`), fold-left (`foldl`) and fold-right (`foldr`). To handle ternary operators such as `foldl`

⁶<http://dream.inf.ed.ac.uk/projects/isascheme/eswa>

and *foldr*, we used another propositional scheme

$$\left(\begin{array}{l} \text{prop-scheme-ternary}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{v}) \equiv \\ \forall w x y z. \mathbf{p}(\mathbf{v}(z, x, y), \mathbf{q}(z, x, y), \mathbf{r}(w, x, z)) = \\ \mathbf{s}(z, \mathbf{t}(y, x, z), \mathbf{u}(z, y, w)) \end{array} \right) \quad (10)$$

with the following closed terms

$$X_P = \left\{ \begin{array}{l} (\lambda x y z. x), (\lambda x y z. y), (\lambda x y z. z), (\lambda x y z. []), (\lambda x y z. \#) \\ (\lambda x. @), (\lambda x y z. @), (\lambda x y. y @ x), \text{foldl}, \text{foldr} \end{array} \right\}$$

IsaScheme produced a total of 4 theorems for the theory of lists producing 2 out of 4 theorems about append, fold-left and fold-right included in Isabelle's libraries. Associativity of append $(x @ y) @ z = x @ (y @ z)$ and $x @ [] = x$ were not synthesised by IsaScheme. Interestingly, append was redefined in terms of fold-right and the list constructor $\#$ by $x @ z = \text{foldr}(\#, x, z)$. This caused associativity of append to be subsumed by the more general version $\text{foldr}(z, \text{foldr}(\#, x, y), w) = \text{foldr}(z, x, \text{foldr}(z, y, w))$ and $x @ [] = x$ to be subsumed by $\text{foldr}(\#, z, []) = z$. In this theory, there was a rather high number of unfalsified and unproved conjectures (129). The reason for this is that the type information in variables for these conjectures is more complex than Quickcheck and Nitpick can manage. A small random sample of these conjectures was taken, and in each case, a counterexample was quite easily found by hand. For instance, one of these unfalsified conjectures was $\text{foldr}(z, x, \#) = \#$, which can be falsified with the witness $\sigma = \{z \mapsto (\lambda a b c d. d), x \mapsto [a]\}$.

In total, IsaScheme produced 13 theorems for the two theories for lists analysed producing all 9 theorems about append, list reverse, map, right-fold

and left-fold included in Isabelle’s libraries. This gives a precision of 70% and a recall of 100%. Table 3 summarises the statistics for the theories analysed.

Precision-Recall	70%-46%	77%-100%	50%-50%
Constructors	Z, S	Z, S, N, C	N, C
Functions Symbols	+, *, ^	A, R, M, L	A, FL, FR
Elapsed Time (s)	1663	1986	8726
Conjectures Synthesised	78957	47142	13213
Conjectures Filtered	78934	47133	13209
Proved-Not Proved	23-0	9-0	4-129

Table 3: Precision and recall analysis with Isabelle’s theory library as reference. The constructors are 0, *suc*, [] and # with labels Z, S, N and C respectively. The functions are +, *, ^, @, *rev*, *len*, *map*, *foldl* and *foldr* with labels +, *, ^, A, R, L, M, FL and FR respectively.

10. Related Work

Other than IsaScheme, Theorema is the only system performing the exploration of mathematical theories based on schemes [9]. However, the user needs to perform all schematic substitutions manually as Theorema does not instantiate the schemes automatically from a set of terms. The user also needs to conduct the proof obligations interactively [10]. Another important difference is that ensuring the soundness of definitions is left to the user in Theorema. In IsaScheme, which uses Isabelle’s LCF-methodology, definitions are sound by construction [14].

The MATHsAiD program was intended for use of research mathematicians and was designed to produce interesting theorems from the mathematician’s point of view [6]. MATHsAiD starts with an axiomatic description of a theory; hypotheses and terms of interest are then generated, forward reasoning is then applied to produce logical consequences of the hypotheses and then a filtering process is carried out according to a number of interestingness measures. MATHsAiD has been applied to the naturals, set theory and group theory.

Like IsaScheme, IsaCoSy is a theory exploration system for Isabelle/IsaPlanner [15]. It generates conjectures in a bottom-up fashion from the signature of an inductive theory. The synthesis process is accompanied by automatic counter-example checking and a prove attempt in case no counter-example is found by Quickcheck. All theorems found are then used as constraints for the synthesis process; synthesis generates only irreducible terms w.r.t. a rewrite system defined by the discovered theorems. The main difference between IsaScheme and IsaCoSy is that IsaCoSy considers all (irreducible) terms as candidate conjectures where IsaScheme considers only a restricted set (modulo \mathcal{R}) specified by the schemes. This restricted set of conjectures avoids the need for a sophisticated constraint language. The main advance made by IsaScheme is the use of Knuth-Bendix completion and termination checking to orient the resulting equational theorems to form a rewrite system. The empirical results show that for the theory of lists, these rewrite systems result in fewer theorems that prove all of the theorems in the theory produced by IsaCoSy.

HR is a theory exploration system which uses an example driven approach

for theory formation [5]. It uses MACE to build models from examples and also to identify counter-examples. The resolution prover Otter is used for the proof obligations. The process of concept invention is carried out from old concepts starting with the concepts provided by MACE at the initial stage. These concepts, stored as data-tables of examples rather than definitions, are passed through a set of production rules whose purpose is to manipulate and generate new data-tables, thus generating new concepts. The conjecture synthesis process is built on top of concept formation. HR takes the concepts obtained by the production rules and forms conjectures about them. There are different types of conjectures HR can make, e.g. *equivalence* conjectures which amounts to finding two concepts and stating that their definitions are equivalent, *implication* conjectures are statements relating two concepts by stating that the first is a specialisation of the second (all examples of the first will be examples of the second), etc. HR has been applied to the naturals, group theory and graph theory.

11. Limitations and Future Work

An important aspect of every theory exploration system is its applicability across different mathematical theories. The scheme-based approach used by IsaScheme, provides a generic mechanism for the exploration of any mathematical theory where the symbols (or closed terms built from them) in the theory's signature and the variables within the schemes could be unified. However, it has only been evaluated thus far on two domains. A hypothesis behind the scheme based approach is that only a few schemes are needed to generate most mathematical concepts. Thus further evaluation is an impor-

tant direction for future work.

Theory exploration leads to a substantial number of instantiations that needs to be processed (see section 5). This is particularly true when there are large numbers of constructors and function symbols. This is partially mediated with the lemmata discovered during the exploration of the theory. Nevertheless, this could be improved by also exploiting the intermediate lemmata needed to finish the proofs, e.g. with the lemma calculation critic used in rippling [21].

Another limitation is that termination (and thus confluence) of rewrite systems is in general undecidable and requires sophisticated technology to solve interesting cases. This problem is aggravated with rewrite systems with a large number of rewrite rules. In this situation, termination checking demanded by definition 6 would benefit from modular properties of rewrite systems such as *hierarchical termination*[22].

At the present the user has to provide the terms that are used to instantiate schemes. Future work is to automate the construction of these terms. Moreover, anti-unification might provide a way to also automate the generation of schemes.

As observed in the evaluation, there was low precision of invented definitions. This suggests the need for strategies to assess the relevance of definitions and discard uninteresting ones.

12. Conclusion

We have implemented the proposed scheme-based approach to mathematical theory exploration in Isabelle/HOL for the generation of conjectures

and definitions. This interpretation is used to describe how the instantiation process of schemes can be automated. We have also described how we can make productive use of normalisation in two ways: first to improve proof automation by maintaining a terminating and potentially convergent rewrite system and second to avoid numerous redundancies inherent in most theory exploration systems.

We performed a precision / recall analysis with Isabelle’s libraries to evaluate the quality of the theorems and definitions found by our method and implementation. IsaScheme produces many interesting theorems resulting in high precision of 70% for the natural numbers and 70% for the theory of lists. IsaScheme finds all the theorems for the theory of lists resulting in a recall of 100% but it does not find all the theorems for the naturals obtaining only 46%. Surprisingly, 16 out of 19 theorems not found were subsumed by the theorems IsaScheme synthesised and the rest fell out of the scope of the propositional scheme used. IsaScheme produced all the definitions for the theory of naturals after 3 exploration rounds producing a recall of 100%. However, IsaScheme produced a low recall of definitions for the theory of lists obtaining of only 21%. The culprit was the definitional scheme used as it was only able to synthesise binary functions and not unary or ternary ones. IsaScheme produced a low precision for definitions in both theories obtaining 8% for the natural numbers and 25% for the theory of lists. Strategies to assess the relevance of definitions is left as future work.

Acknowledgments.

This work has been supported by Universidad Politécnica de San Luis Potosí, SEP-PROMEP, University of Edinburgh, the Edinburgh Compute and

Data Facility, the RISC-Linz Transnational Access Programme (No. 026133), and EPSRC grants EP/F033559/1 and EP/E005713/1.

Appendix A. Datatypes and Definition of Functions

Appendix A.1. Natural Numbers

datatype $nat = 0 \mid suc\ nat$

Addition ($+ :: nat \rightarrow nat \rightarrow nat$)

$$\begin{aligned} 0 * y &= 0 \\ suc(x) * y &= y + (x * y) \end{aligned}$$

Multiplication ($* :: nat \rightarrow nat \rightarrow nat$)

$$\begin{aligned} 0 * y &= 0 \\ suc(x) * y &= y + (x * y) \end{aligned}$$

Exponentiation ($\wedge :: nat \rightarrow nat \rightarrow nat$)

$$\begin{aligned} 0 \wedge y &= suc(0) \\ suc(x) \wedge y &= y * (x \wedge y) \end{aligned}$$

Appendix A.2. Lists

datatype $'a\ list = [] \mid \#\ of\ suc\ nat$

Append ($@ :: 'a\ list \rightarrow 'a\ list \rightarrow 'a\ list$)

$$\begin{aligned} [] @ l &= l \\ (h \# t) @ l &= h \# (t @ l) \end{aligned}$$

List reverse ($rev :: 'a\ list \rightarrow 'a\ list$)

$$\begin{aligned} rev([]) &= [] \\ rev(h\#t) &= rev(t)\ @\ (h\#[]) \end{aligned}$$

Map ($map :: ('a \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b\ list$)

$$\begin{aligned} map(f, []) &= [] \\ map(f, a\#b) &= f(a)\#map(f, b) \end{aligned}$$

Length ($len :: 'a\ list \rightarrow nat$)

$$\begin{aligned} len([]) &= 0 \\ len(h\#t) &= suc(len(t)) \end{aligned}$$

Fold-left ($foldl :: ('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b\ list \rightarrow 'a$)

$$\begin{aligned} foldl(f, a, []) &= a \\ foldl(f, a, x\#xs) &= foldl(f, f(a, x), xs) \end{aligned}$$

Fold-right ($foldr :: ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b \rightarrow 'b$)

$$\begin{aligned} foldr(f, [], a) &= a \\ foldr(f, x\#xs, a) &= f(x, foldr(f, xs, a)) \end{aligned}$$

Binary head ($hd :: 'a\ list \rightarrow 'a \rightarrow 'a$)

$$\begin{aligned} hd([], y) &= y \\ hd(z\#x, y) &= z \end{aligned}$$

Binary last ($last :: 'a\ list \rightarrow 'a \rightarrow 'a$)

$$\begin{aligned} last([], y) &= y \\ last(z\#x, y) &= last(x, z) \end{aligned}$$

Tail recursive reverse ($qrev :: 'a\ list \rightarrow 'a\ list \rightarrow 'a\ list$)

$$\begin{aligned}qrev([], y) &= y \\ qrev(z\#x, y) &= qrev(x, z\#y)\end{aligned}$$

Replicate ($replicate :: 'a\ list \rightarrow 'b \rightarrow 'b\ list$)

$$\begin{aligned}replicate([], y) &= y \\ replicate(z\#x, y) &= y\#replicate(x, y)\end{aligned}$$

Appendix B. Theorems Found

Appendix B.1. Natural Numbers

Appendix B.2. Lists

References

- [1] M. Johansson, Automated Discovery of Inductive Lemmas, Ph.D. thesis, School of Informatics, University of Edinburgh (2009).
- [2] G. Sutcliffe, Y. Gao, S. Colton, A Grand Challenge of Theorem Discovery (June 2003).
URL citeseer.ist.psu.edu/691846.html
- [3] A. Craciun, Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory, Ph.D. thesis, Research Institute for Symbolic Computation (RISC) (2008).
- [4] D. B. Lenat, AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search, in: Knowledge-based systems in artificial intelligence, McGraw Hill, 1982, pp. 833–842, also available from Stanford as TechReport AIM 286.

No.	Theorem	$\in \mathcal{R}$
1	$x + 0 = x$	✓
2	$x * 0 = 0$	✓
3	$suc(0) \hat{=} x = suc(0)$	✓
4	$y + suc(z) = suc(y + z)$	✓
5	$y + x = x + y$	✗
6*	$x + (z + y) = x + (y + z)$	✗
7	$(x + y) + z = x + (y + z)$	✓
8	$y + (x + z) = x + (y + z)$	✗
9	$(x + y) * z = x * z + y * z$	✓
10	$x * suc(z) = x + x * z$	✓
11	$x * (y + z) = x * y + x * z$	✓
12	$y * x = x * y$	✗
13*	$x * (z * y) = x * (y * z)$	✗
14*	$y + z * y = y + y * z$	✗
15*	$y * x + y * z = x * y + z * y$	✗
16*	$z + (y + y * z) = y + (z + z * y)$	✗
17*	$x + (y * x + (z + y * z)) = x + (x * y + (z + z * y))$	✗
18	$(x * y) * z = x * (y * z)$	✓
19	$y * (x * z) = x * (y * z)$	✗
20	$(x * y) \hat{=} z = x \hat{=} z * y \hat{=} z$	✓
21	$x \hat{=} (y + z) = x \hat{=} y * x \hat{=} z$	✓
22*	$x \hat{=} (z * y) = x \hat{=} (y * z)$	✗
23	$(x \hat{=} y) \hat{=} z = x \hat{=} (y * z)$	✓

Table B.4: Theorems found about addition, multiplication and exponentiation in the theory of natural numbers. Theorems marked with * are not included in Isabelles nat library. The third column indicates whether the theorem is included in the TRS \mathcal{R} .

No.	Theorem	$\in \mathcal{R}$
1	$x@[] = x$	✓
2	$len(rev(x)) = len(x)$	✓
3*	$len(map(y, x)) = len(x)$	✓
4	$(x @ y) @ z = x @ (y @ z)$	✓
5	$rev(y @ x) = rev(x) @ rev(y)$	✓
6	$rev(rev(x)) = x$	✓
7	$map(z, x @ y) = map(z, x @ map(z, y))$	✓
8	$rev(map(x, z)) = map(x, rev(z))$	✓
9*	$len(z @ x) = len(x @ z)$	✗

Table B.5: Theorems found about append ($@$), rev (rev), map (map) and length (len) in the theory of natural numbers. Theorems marked with * are not included in Isabelles list library. The third column indicates whether the theorem is included in the TRS \mathcal{R} .

No.	Theorem	$\in \mathcal{R}$
1*	$foldr(\#, z, []) = z$	✓
2*	$x@z = foldr(\#, x, z)$	✓
3	$foldl(z, foldl(z, x, y), w) = foldl(z, x, foldr(\#, y, w))$	✗
4	$foldr(z, foldr(\#, x, y), w) = foldr(z, x, foldr(z, y, w))$	✓

Table B.6: Theorems found about append ($@$), foldl and foldr in the theory of lists. Theorems marked with * are not included in Isabelles list library. The third column indicates whether the theorem is included in the TRS \mathcal{R} .

- [5] S. Colton, Automated Theory Formation in Pure Mathematics, Ph.D. thesis, Division of Informatics, University of Edinburgh (2001).
- [6] R. McCasland, A. Bundy, P. F. Smith, Ascertaining Mathematical Theorems, Electr. Notes Theor. Comput. Sci. 151 (1) (2006) 21–38.

- [7] D. B. Lenat, Automated Theory Formation in Mathematics, in: IJ-CAI'77: Proceedings of the 5th international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1977, pp. 833–842.
- [8] B. Buchberger, Algorithm-Supported Mathematical Theory Exploration: A Personal View and Strategy (2004).
- [9] B. Buchberger, A. Craciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, Theorema: Towards Computer-aided Mathematical Theory Exploration, *J. Applied Logic* 4 (4) (2006) 470–504.
- [10] A. Craciun, M. Hodorog, Decompositions of Natural Numbers: From a Case Study in Mathematical Theory Exploration, *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2007) 41–47.
- [11] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle's Logics: HOL (2000).
- [12] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.
- [13] L. Dixon, J. D. Fleuriot, IsaPlanner: A Prototype Proof Planner in Isabelle, in: *Proceedings of CADE'03*, Vol. 2741 of LNCS, 2003, pp. 279–283.
- [14] A. Krauss, Automating Recursive Definitions and Termination Proofs in

- Higher-Order Logic, Ph.D. thesis, Dept. of Informatics, T. U. München (2009).
- [15] M. Johansson, L. Dixon, A. Bundy, Conjecture Synthesis for Inductive Theories, *Journal of Automated Reasoning*(To appear).
- [16] J. Giesl, P. Schneider-kamp, R. Thiemann, AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework, in: *In Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, Springer-Verlag, 2006, pp. 281–286.
- [17] I. Wehrman, A. Stump, E. Westbrook, Slothrop: KnuthBendix Completion with a Modern Termination Checker, in: *Webster University, St. Louis, Missouri M.Sc. Computer Science, Washington University, 2006*, pp. 268–279.
- [18] A. Bundy, The Automation of Proof by Mathematical Induction, in: *A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Volume 1, Elsevier, 2001*, pp. 845–911.
- [19] S. Berghofer, T. Nipkow, Random Testing in Isabelle/HOL, in: *SEFM, 2004*, pp. 230–239.
- [20] J. C. Blanchette, T. Nipkow, Nitpick: A Counterexample Generator for Higher-order Logic based on a Relational Model Finder, *Tech. rep., In TAP 2009: Short Papers, ETH (2009)*.
- [21] A. Bundy, D. Basin, D. Hutter, A. Ireland, Rippling: Meta-level Guidance for Mathematical Reasoning, *Vol. 56 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2005*.

- [22] N. Dershowitz, Hierarchical Termination, in: CTRS '94: Proceedings of the 4th International Workshop on Conditional and Typed Rewriting Systems, Springer-Verlag, London, UK, 1995, pp. 89–105.